Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$*

René Thiemann

February 23, 2021

Abstract

We apply data refinement to implement the real numbers, where we support all numbers in the field extension $\mathbb{Q}[\sqrt{b}]$, i.e., all numbers of the form $p + q\sqrt{b}$ for rational numbers p and q and some fixed natural number b. To this end, we also developed algorithms to precisely compute roots of a rational number, and to perform a factorization of natural numbers which eliminates duplicate prime factors.

Our results have been used to certify termination proofs which involve polynomial interpretations over the reals.

Contents

1 Introduction

It has been shown that polynomial interpretations over the reals are strictly more powerful for termination proving than polynomial interpretations over the rationals. To this end, also automated termination prover started to generate such interpretations. [3, 4, 5, 7, 8]. However, for all current implementations, only reals of the form $p + q \cdot \sqrt{b}$ are generated where b is some fixed natural number and p and q may be arbitrary rationals, i.e., we get numbers within $\mathbb{Q}[\sqrt{b}]$.

*This research is supported by FWF (Austrian Science Fund) project P22767-N13.

To support these termination proofs in our certifier CeTA [6], we therefore required executable functions on $\mathbb{Q}[\sqrt{b}]$, which can then be used as an implementation type for the reals. Here, we used ideas from [1, 2] to provide a sufficiently powerful partial implementations via data refinement.

2 Auxiliary lemmas which might be moved into the Isabelle distribution.

theory *Real-Impl-Auxiliary*
imports
HOL−Computational-Algebra.Primes
begin

lemma *multiplicity-prime*:
assumes *p*: *prime* (*i* :: *nat*) **and** *ji*: $j \neq i$
shows *multiplicity j i* = *0*
using *assms*
by (*metis dvd-refl prime-nat-iff multiplicity-eq-zero-iff*
multiplicity-unit-left multiplicity-zero)

end

3 Prime products

theory *Prime-Product*
imports
Real-Impl-Auxiliary
Sqrt-Babylonian.Sqrt-Babylonian
begin

Prime products are natural numbers where no prime factor occurs more than once.

definition *prime-product*
where *prime-product* (*n* :: *nat*) = (∀ *p*. *prime p* ⟶ *multiplicity p n* ≤ *1*)

The main property is that whenever b_1 and b_2 are different prime products, then $p_1 + q_1\sqrt{b_1} = p_2 + q_2\sqrt{b_2}$ implies $(p_1, q_1, b_1) = (p_2, q_2, b_2)$ for all rational numbers p_1, q_1, p_2, q_2. This is the key property to uniquely represent numbers in $\mathbb{Q}[\sqrt{b}]$ by triples. In the following we develop an algorithm to decompose any natural number n into $n = s^2 \cdot p$ for some s and prime product p.

function *prime-product-factor-main* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* × *nat* **where**
prime-product-factor-main factor-sq factor-pr limit n i =
(*if i* ≤ *limit* ∧ *i* ≥ *2 then*
(*if i dvd n*

```
        then (let n' = n div i in
          (if i dvd n' then
             let n'' = n' div i in
            prime-product-factor-main (factor-sq * i) factor-pr (nat (root-nat-floor
3 n'')) n'' i
           else
            (case sqrt-nat n' of
              Cons sn - ⇒ (factor-sq * sn, factor-pr * i)
                | [] ⇒ prime-product-factor-main factor-sq (factor-pr * i) (nat
(root-nat-floor 3 n')) n' (Suc i)
            )
          )
        )
       else
        prime-product-factor-main factor-sq factor-pr limit n (Suc i))
     else
      (factor-sq, factor-pr * n)) by pat-completeness auto

termination
proof −
  let ?m1 = λ (factor-sq :: nat,factor-pr :: nat,limit :: nat,n :: nat,i :: nat). n
  let ?m2 = λ (factor-sq,factor-pr,limit,n,i). (Suc limit − i)
  {
    fix i
    have 2 ≤ i ⟹ Suc 0 < i * i using one-less-mult[of i i] by auto
  } note * = this
  show ?thesis
    using wf-measures [of [?m1, ?m2]]
    by rule (auto simp add: * elim!: dvdE split: if-splits)
qed

lemma prime-product-factor-main: assumes ¬ (∃ s. s * s = n)
  and limit = nat (root-nat-floor 3 n)
  and m = factor-sq * factor-sq * factor-pr * n
  and prime-product-factor-main factor-sq factor-pr limit n i = (sq, p)
  and i ≥ 2
  and ⋀ j. j ≥ 2 ⟹ j < i ⟹ ¬ j dvd n
  and ⋀ j. prime j ⟹ j < i ⟹ multiplicity j factor-pr ≤ 1
  and ⋀ j. prime j ⟹ j ≥ i ⟹ multiplicity j factor-pr = 0
  and factor-pr > 0
  shows m = sq * sq * p ∧ prime-product p
  using assms
proof (induct factor-sq factor-pr limit n i rule: prime-product-factor-main.induct)
  case (1 factor-sq factor-pr limit n i)
  note IH = 1(1−3)
  note prems = 1(4−)
  note simp = prems(4)[unfolded prime-product-factor-main.simps[of factor-sq fac-
tor-pr limit n i]]
  show ?case
```

```
  proof (cases i ≤ limit)
    case True note i = this
     with prems(5) have cond: i ≤ limit ∧ i ≥ 2 and *: (i ≤ limit ∧ i ≥ 2) =
True by blast+
    note IH = IH[OF cond]
    note simp = simp[unfolded * if-True]
    show ?thesis
    proof (cases i dvd n)
      case False
      hence *: (i dvd n) = False by simp
      note simp = simp[unfolded * if-False]
      note IH = IH(3)[OF False prems(1–3) simp]
      show ?thesis
      proof (rule IH)
        fix j
        assume 2: 2 ≤ j and j: j < Suc i
        from prems(6)[OF 2] j False
        show ¬ j dvd n by (cases j = i, auto)
      next
        fix j :: nat
        assume j: j < Suc i prime j
        with prems(7–8)[of j]
        show multiplicity j factor-pr ≤ 1 by (cases j = i, auto)
      qed (insert prems(8–9) cond, auto)
    next
      case True note mod = this
      hence (i dvd n) = True by simp
      note simp = simp[unfolded this if-True Let-def]
      note IH = IH(1,2)[OF True refl]
      show ?thesis
      proof (cases i dvd n div i)
        case True
        hence *: (i dvd n div i) = True by auto
        define n' where n' = n div i div i
       from mod True have n: n = n' * i * i by (auto simp: n'-def dvd-eq-mod-eq-0)
        note simp = simp[unfolded * if-True split]
        note IH = IH(1)[OF True refl - refl - simp prems(5) - prems(7–9)]
        show ?thesis
        proof (rule IH)
          show m = factor-sq * i * (factor-sq * i) * factor-pr * (n div i div i)
            unfolding prems(3) n'-def[symmetric]
            unfolding n by (auto simp: field-simps)
        next
          fix j
          assume 2 ≤ j j < i
          from prems(6)[OF this] have ¬ j dvd n by auto
          thus ¬ j dvd n div i div i
            by (metis dvd-mult n n'-def mult.commute)
        next
```

```
    show ¬ (∃ s. s * s = n div i div i)
    proof
      assume ∃ s. s * s = n div i div i
      then obtain s where s * s = n div i div i by auto
      hence (s * i) * (s * i) = n unfolding n by auto
      with prems(1) show False by blast
    qed
  qed
next
  case False
  define n' where n' = n div i
  from mod True have n: n = n' * i by (auto simp: n'-def dvd-eq-mod-eq-0)
  have prime: prime i
    unfolding prime-nat-iff
  proof (intro conjI allI impI)
    fix m
    assume m: m dvd i
    hence m dvd n unfolding n by auto
    with prems(6)[of m] have choice: m ≤ 1 ∨ m ≥ i by arith
    from m prems(5) have m > 0
      by (metis dvd-0-left-iff le0 le-antisym neq0-conv zero-neq-numeral)
    with choice have choice: m = 1 ∨ m ≥ i by arith
    from m prems(5) have m ≤ i
      by (metis False div-by-0 dvd-refl dvd-imp-le gr0I)
    with choice
    show m = 1 ∨ m = i by auto
  qed (insert prems(5), auto)
  from False have (i dvd n div i) = False by auto
  note simp = simp[unfolded this if-False]
  note IH = IH(2)[OF False - - refl]
  from prime have i > 0 by (simp add: prime-gt-0-nat)

  show ?thesis
  proof (cases sqrt-nat (n div i))
    case (Cons s)
    note simp = simp[unfolded Cons list.simps]
    hence sq: sq = factor-sq * s and p: p = factor-pr * i by auto
    from arg-cong[OF Cons, of set] have s: s * s = n div i by auto
    have pp: prime-product (factor-pr * i)
      unfolding prime-product-def
    proof safe
      fix m :: nat assume m: prime m
      consider i < m | i > m | i = m by force
      thus multiplicity m (factor-pr * i) ≤ 1
        by cases (insert prems(7)[of m] prems(8)[of m] prems(9) ‹i > 0› prime
m,
            simp-all add: multiplicity-prime prime-elem-multiplicity-mult-distrib)
    qed
    show ?thesis unfolding sq p prems(3) n unfolding n'-def s[symmetric]
```

```
    using pp by auto
  next
    case Nil
    note simp = simp[unfolded Nil list.simps]
    from arg-cong[OF Nil, of set] have ¬ (∃ x. x * x = n div i) by simp
    note IH = IH[OF Nil this  - simp]
    show ?thesis
    proof (rule IH)
      show m = factor-sq * factor-sq * (factor-pr * i) * (n div i)
        unfolding prems(3) n by auto
    next
      fix j
      assume *: 2 ≤ j j < Suc i
      show ¬ j dvd n div i
      proof
        assume j: j dvd n div i
        with False have j ≠ i by auto
        with * have 2 ≤ j j < i by auto
        from prems(6)[OF this] j
        show False unfolding n
          by (metis dvd-mult n n'-def mult.commute)
      qed
    next
      fix j :: nat
      assume Suc i ≤ j and j-prime: prime j
      hence ij: i ≤ j and j: j ≠ i by auto
      have 0: multiplicity j i = 0 using prime j by (rule multiplicity-prime)
      show multiplicity j (factor-pr * i) = 0
        unfolding prems(8)[OF j-prime ij] 0
        using prime j-prime j ‹ 0 < factor-pr› ‹multiplicity j factor-pr = 0›
         by (subst prime-elem-multiplicity-mult-distrib) (auto simp: multiplic-
ity-prime)
    next
      fix j
      assume j < Suc i and j-prime: prime j
      hence j < i ∨ j = i by auto
      thus multiplicity j (factor-pr * i) ≤ 1
      proof
        assume j = i
        with prems(8)[of i] prime j-prime ‹0 < factor-pr› show ?thesis
          by (subst prime-elem-multiplicity-mult-distrib) auto
      next
        assume ji: j < i
        hence j ≠ i by auto
        from prems(7)[OF j-prime ji] multiplicity-prime[OF prime this]
            prime j-prime ‹0 < factor-pr›
        show ?thesis by (subst prime-elem-multiplicity-mult-distrib) auto
      qed
    qed (insert prems(5,9), auto)
```

```
      qed
    qed
  qed
 next
  case False
  hence (i ≤ limit ∧ i ≥ 2) = False by auto
  note simp = simp[unfolded this if-False]
  hence sq: sq = factor-sq and p: p = factor-pr * n by auto
  show ?thesis
  proof
   show m = sq * sq * p unfolding sq p prems(3) by simp
   show prime-product p unfolding prime-product-def
   proof safe
    fix m :: nat assume m: prime m
    from prems(1) have n1: n > 1 by (cases n, auto, case-tac nat, auto)
    hence n0: n > 0 by auto
    have i > limit using False by auto
     from this[unfolded prems(2)] have less: int i ≥ root-nat-floor 3 n + 1 by
auto
     have int n < (root-nat-floor 3 n + 1) ^ 3 by (rule root-nat-floor-upper,
auto)
    also have ... ≤ int i ^ 3 by (rule power-mono[OF less, of 3], auto)
    finally have n-i3: n < i ^ 3
     by (metis of-nat-less-iff of-nat-power [symmetric])
    {
     fix m
     assume m: prime m multiplicity m n > 0
     hence mp: m ∈ prime-factors n
      by (auto simp: prime-factors-multiplicity)
     hence md: m dvd n
      by auto
     then obtain k where n: n = m * k ..
     from mp have pm: prime m by auto
     hence m2: m ≥ 2 and m0: m > 0 by (auto simp: prime-nat-iff)
     from prems(6)[OF m2] md have mi: m ≥ i by force
     {
      assume multiplicity m n ≠ 1
      with m have ∃ k. multiplicity m n = 2 + k by presburger
      then obtain j where mult: multiplicity m n = 2 + j ..
      from n0 n have k: k > 0 by auto
      from mult m0 k n m have multiplicity m k > 0
       by (auto simp: prime-elem-multiplicity-mult-distrib)
      with m have mp: m ∈ prime-factors k
       by (auto simp: prime-factors-multiplicity)
      hence md: m dvd k by (auto simp: k)
      then obtain l where kml: k = m * l ..
      note n = n[unfolded kml]
      from n have l dvd n by auto
      with prems(6)[of l] have l ≤ 1 ∨ l ≥ i by arith
```

```
        with n n0 have l: l = 1 ∨ l ≥ i by auto
        from n prems(1) have l ≠ 1 by auto
        with l have l: l ≥ i by auto
        from mult-le-mono[OF mult-le-mono[OF mi mi] l]
        have n ≥ i^3 unfolding n by (auto simp: power3-eq-cube)
        with n-i3 have False by auto
      }
      with mi m
      have multiplicity m n = 1 ∧ m ≥ i by auto
    } note n = this
    have multiplicity m p = multiplicity m factor-pr + multiplicity m n
      unfolding p using prems(1,9) m ‹n > 0›
      by (auto simp: prime-elem-multiplicity-mult-distrib)
    also have ... ≤ 1
    proof (cases m < i)
      case True
      from prems(7)[of m] n[of m] True m show ?thesis by force
    next
      case False
      hence m ≥ i by auto
      from prems(8)[OF m(1) this] n[of m] m show ?thesis by force
    qed
    finally show multiplicity m p ≤ 1 .
  qed
 qed
qed


definition prime-product-factor :: nat ⇒ nat × nat where
 prime-product-factor n = (case sqrt-nat n of
   (Cons s -) ⇒ (s,1)
  | [] ⇒ prime-product-factor-main 1 1 (nat (root-nat-floor 3 n)) n 2)

lemma prime-product-one[simp, intro]: prime-product 1
 unfolding prime-product-def multiplicity-one-nat by auto

lemma prime-product-factor: assumes pf: prime-product-factor n = (sq,p)
 shows n = sq * sq * p ∧ prime-product p
proof (cases sqrt-nat n)
 case (Cons s)
 from pf[unfolded prime-product-factor-def Cons] arg-cong[OF Cons, of set]
   prime-product-one
 show ?thesis by auto
next
 case Nil
 from arg-cong[OF Nil, of set] have nsq: ¬ (∃ s. s * s = n) by auto
 show ?thesis
   by (rule prime-product-factor-main[OF nsq refl, of - 1 1 2], unfold multiplic-
```

ity-one,
insert pf[*unfolded prime-product-factor-def Nil*], *auto*)
qed

end

4 A representation of real numbers via triples

theory *Real-Impl*
imports
Sqrt-Babylonian.Sqrt-Babylonian
begin

We represent real numbers of the form $p + q \cdot \sqrt{b}$ for $p, q \in \mathbb{Q}$, $n \in \mathbb{N}$ by triples (p, q, b). However, we require the invariant that $\sqrt{b}$ is irrational. Most binary operations are implemented via partial functions where the common the restriction is that the numbers b in both triples have to be identical. So, we support addition of $\sqrt{2} + \sqrt{2}$, but not $\sqrt{2} + \sqrt{3}$.

The set of natural numbers whose sqrt is irrational

definition *sqrt-irrat* = { *q* :: *nat*. ¬ (∃ *p*. *p* ∗ *p* = *rat-of-nat q*)}

lemma *sqrt-irrat*: **assumes** *choice*: *q* = *0* ∨ *b* ∈ *sqrt-irrat*
and *eq*: *real-of-rat p* + *real-of-rat q* ∗ *sqrt* (*of-nat b*) = *0*
shows *q* = *0*
using *choice*
proof (*cases q* = *0*)
case *False*
with *choice* **have** *sqrt-irrat*: *b* ∈ *sqrt-irrat* **by** *blast*
from *eq* **have** *real-of-rat q* ∗ *sqrt* (*of-nat b*) = *real-of-rat* (− *p*)
by (*auto simp*: *of-rat-minus*)
then obtain *p* **where** *real-of-rat q* ∗ *sqrt* (*of-nat b*) = *real-of-rat p* **by** *blast*
from *arg-cong*[*OF this*, *of* λ *x*. *x* ∗ *x*] **have** *real-of-rat* (*q* ∗ *q*) ∗ (*sqrt* (*of-nat b*)
∗ *sqrt* (*of-nat b*)) =
real-of-rat (*p* ∗ *p*) **by** (*auto simp*: *field-simps of-rat-mult*)
also have *sqrt* (*of-nat b*) ∗ *sqrt* (*of-nat b*) = *of-nat b* **by** *simp*
finally have *real-of-rat* (*q* ∗ *q* ∗ *rat-of-nat b*) = *real-of-rat* (*p* ∗ *p*) **by** (*auto simp*:
of-rat-mult)
hence *q* ∗ *q* ∗ (*rat-of-nat b*) = *p* ∗ *p* **by** *auto*
from *arg-cong*[*OF this*, *of* λ *x*. *x* / (*q* ∗ *q*)]
have (*p* / *q*) ∗ (*p* / *q*) = *rat-of-nat b* **using** *False* **by** (*auto simp*: *field-simps*)
with *sqrt-irrat* **show** *?thesis* **unfolding** *sqrt-irrat-def* **by** *blast*
qed

To represent numbers of the form $p + q \cdot \sqrt{b}$, use mini algebraic numbers, i.e., triples (p, q, b) with irrational $\sqrt{b}$.

typedef *mini-alg* =
{(*p*,*q*,*b*) | (*p* :: *rat*) (*q* :: *rat*) (*b* :: *nat*).
q = *0* ∨ *b* ∈ *sqrt-irrat*}

by *auto*

setup-lifting *type-definition-mini-alg*

lift-definition *real-of* :: *mini-alg* ⇒ *real* **is**
λ (*p*,*q*,*b*). *of-rat p* + *of-rat q* ∗ *sqrt* (*of-nat b*) **.**

lift-definition *ma-of-rat* :: *rat* ⇒ *mini-alg* **is** λ *x*. (*x*,*0*,*0*) **by** *auto*

lift-definition *ma-rat* :: *mini-alg* ⇒ *rat* **is** *fst* **.**
lift-definition *ma-base* :: *mini-alg* ⇒ *nat* **is** *snd o snd* **.**
lift-definition *ma-coeff* :: *mini-alg* ⇒ *rat* **is** *fst o snd* **.**

lift-definition *ma-uminus* :: *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*). (− *p1*, − *q1*, *b1*) **by** *auto*

lift-definition *ma-compatible* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *q1* = *0* ∨ *q2* = *0* ∨ *b1* = *b2* **.**

definition *ma-normalize* :: *rat* × *rat* × *nat* ⇒ *rat* × *rat* × *nat* **where**
ma-normalize x ≡ *case x of* (*a*,*b*,*c*) ⇒ *if b* = *0 then* (*a*,*0*,*0*) *else* (*a*,*b*,*c*)

lemma *ma-normalize-case*[*simp*]: (*case ma-normalize r of* (*a*,*b*,*c*) ⇒ *real-of-rat a* + *real-of-rat b* ∗ *sqrt* (*of-nat c*))
= (*case r of* (*a*,*b*,*c*) ⇒ *real-of-rat a* + *real-of-rat b* ∗ *sqrt* (*of-nat c*))
by (*cases r*, *auto simp*: *ma-normalize-def*)

lift-definition *ma-plus* :: *mini-alg* ⇒ *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *if q1* = *0 then*
(*p1* + *p2*, *q2*, *b2*) *else ma-normalize* (*p1* + *p2*, *q1* + *q2*, *b1*) **by** (*auto simp*: *ma-normalize-def*)

lift-definition *ma-times* :: *mini-alg* ⇒ *mini-alg* ⇒ *mini-alg* **is**
λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*). *if q1* = *0 then*
ma-normalize (*p1*∗*p2*, *p1*∗*q2*, *b2*) *else*
ma-normalize (*p1*∗*p2* + *of-nat b2*∗*q1*∗*q2*, *p1*∗*q2* + *q1*∗*p2*, *b1*) **by** (*auto simp*: *ma-normalize-def*)

lift-definition *ma-inverse* :: *mini-alg* ⇒ *mini-alg* **is**
λ (*p*,*q*,*b*). *let d* = *inverse* (*p* ∗ *p* − *of-nat b* ∗ *q* ∗ *q*) *in*
ma-normalize (*p* ∗ *d*, − *q* ∗ *d*, *b*) **by** (*auto simp*: *Let-def ma-normalize-def*)

lift-definition *ma-floor* :: *mini-alg* ⇒ *int* **is**
λ (*p*,*q*,*b*). *case* (*quotient-of p*,*quotient-of q*) *of* ((*z1*,*n1*),(*z2*,*n2*)) ⇒
let z2n1 = *z2* ∗ *n1*; *z1n2* = *z1* ∗ *n2*; *n12* = *n1* ∗ *n2*; *prod* = *z2n1* ∗ *z2n1* ∗ *int b in*
(*z1n2* + (*if z2n1* ≥ *0 then sqrt-int-floor-pos prod else* − *sqrt-int-ceiling-pos prod*)) *div n12* **.**

lift-definition *ma-sqrt* :: *mini-alg* ⇒ *mini-alg* **is**
  λ (*p*,*q*,*b*). *let* (*a*,*b*) = *quotient-of p*; *aa* = *abs* (*a* ∗ *b*) *in*
  case sqrt-int aa of [] ⇒ (*0*,*inverse* (*of-int b*),*nat aa*) | (*Cons s* -) ⇒ (*of-int s* / *of-int b*,*0*,*0*)
proof (*unfold Let-def*)
  fix *prod* :: *rat* × *rat* × *nat*
  obtain *p q b* **where** *prod*: *prod* = (*p*,*q*,*b*) **by** (*cases prod*, *auto*)
  obtain *a b* **where** *p*: *quotient-of p* = (*a*,*b*) **by** *force*
  show ∃ *p q b*. (*case prod of*
                (*p*, *q*, *b*) ⇒
                  case quotient-of p of
                  (*a*, *b*) ⇒
                   (*case sqrt-int* |*a* ∗ *b*| *of* [] ⇒ (*0*, *inverse* (*of-int b*), *nat* |*a* ∗ *b*|)
                    | *s* # *x* ⇒ (*of-int s* / *of-int b*, *0*, *0*))) =
               (*p*, *q*, *b*) ∧
               (*q* = *0* ∨ *b* ∈ *sqrt-irrat*)
  proof (*cases sqrt-int* (*abs* (*a* ∗ *b*)))
    case *Nil*
    from *sqrt-int*[*of abs* (*a* ∗ *b*)] *Nil* **have** *irrat*: ¬ (∃ *y*. *y* ∗ *y* = |*a* ∗ *b*|) **by** *auto*
    have *nat* |*a* ∗ *b*| ∈ *sqrt-irrat*
    proof (*rule ccontr*)
      assume *nat* |*a* ∗ *b*| ∉ *sqrt-irrat*
      then obtain *x* :: *rat*
      where *x* ∗ *x* = *rat-of-nat* (*nat* |*a* ∗ *b*|) **unfolding** *sqrt-irrat-def* **by** *auto*
      hence *x* ∗ *x* = *rat-of-int* |*a* ∗ *b*| **by** *auto*
      from *sqr-rat-of-int*[*OF this*] *irrat* **show** *False* **by** *blast*
    qed
    thus *?thesis* **using** *Nil* **by** (*auto simp*: *prod p*)
  qed (*auto simp*: *prod p Cons*)
qed

lift-definition *ma-equal* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is**
  λ (*p1*,*q1*,*b1*) (*p2*,*q2*,*b2*).
  p1 = *p2* ∧ *q1* = *q2* ∧ (*q1* = *0* ∨ *b1* = *b2*) **.**

lift-definition *ma-ge-0* :: *mini-alg* ⇒ *bool* **is**
  λ (*p*,*q*,*b*). *let bqq* = *of-nat b* ∗ *q* ∗ *q*; *pp* = *p* ∗ *p in*
  0 ≤ *p* ∧ *bqq* ≤ *pp* ∨ *0* ≤ *q* ∧ *pp* ≤ *bqq* **.**

lift-definition *ma-is-rat* :: *mini-alg* ⇒ *bool* **is**
  λ (*p*,*q*,*b*). *q* = *0* **.**

definition *ge-0* :: *real* ⇒ *bool* **where** [*code del*]: *ge-0 x* = (*x* ≥ *0*)

lemma *ma-ge-0*: *ge-0* (*real-of x*) = *ma-ge-0 x*
proof (*transfer*, *unfold Let-def*, *clarsimp*)
  fix *p′ q′* :: *rat* **and** *b′* :: *nat*
  assume *b′*: *q′* = *0* ∨ *b′* ∈ *sqrt-irrat*
  define *b* **where** *b* = *real-of-nat b′*

define p **where** $p = \mathit{real\text{-}of\text{-}rat}\ p'$
define q **where** $q = \mathit{real\text{-}of\text{-}rat}\ q'$
from b' **have** b: $0 \leq b\ q = 0 \vee b' \in \mathit{sqrt\text{-}irrat}$ **unfolding** $\mathit{b\text{-}def}\ \mathit{q\text{-}def}$ **by** auto
define qb **where** $qb = q * \mathit{sqrt}\ b$
from b **have** sqrt: $\mathit{sqrt}\ b \geq 0$ **by** auto
from $b(2)$ **have** disj: $q = 0 \vee b \neq 0$ **unfolding** $\mathit{sqrt\text{-}irrat\text{-}def}\ \mathit{b\text{-}def}$ **by** auto
have bdef: $b = \mathit{real\text{-}of\text{-}rat}\ (\mathit{of\text{-}nat}\ b')$ **unfolding** $\mathit{b\text{-}def}$ **by** auto
have $(0 \leq p + q * \mathit{sqrt}\ b) = (0 \leq p + qb)$ **unfolding** $\mathit{qb\text{-}def}$ **by** simp
also have $\ldots \longleftrightarrow (0 \leq p \wedge \mathit{abs}\ qb \leq \mathit{abs}\ p \vee 0 \leq qb \wedge \mathit{abs}\ p \leq \mathit{abs}\ qb)$ **by** arith
also have $\ldots \longleftrightarrow (0 \leq p \wedge qb * qb \leq p * p \vee 0 \leq qb \wedge p * p \leq qb * qb)$
unfolding $\mathit{abs\text{-}lesseq\text{-}square}$ **..**
also have $qb * qb = b * q * q$ **unfolding** $\mathit{qb\text{-}def}$
using b **by** auto
also have $0 \leq qb \longleftrightarrow 0 \leq q$ **unfolding** $\mathit{qb\text{-}def}$ **using** $\mathit{sqrt}\ \mathit{disj}$
by ($\mathit{metis}\ \mathit{le\text{-}cases}\ \mathit{mult\text{-}eq\text{-}0\text{-}iff}\ \mathit{mult\text{-}nonneg\text{-}nonneg}\ \mathit{mult\text{-}nonpos\text{-}nonneg}\ \mathit{order\text{-}class.order.antisym}\ \mathit{qb\text{-}def}\ \mathit{real\text{-}sqrt\text{-}eq\text{-}zero\text{-}cancel\text{-}iff}$)
also have $(0 \leq p \wedge b * q * q \leq p * p \vee 0 \leq q \wedge p * p \leq b * q * q)$
$\longleftrightarrow (0 \leq p' \wedge \mathit{of\text{-}nat}\ b' * q' * q' \leq p' * p' \vee 0 \leq q' \wedge p' * p' \leq \mathit{of\text{-}nat}\ b' * q' * q')$ **unfolding** $\mathit{qb\text{-}def}$
by ($\mathit{unfold}\ \mathit{bdef}\ \mathit{p\text{-}def}\ \mathit{q\text{-}def}\ \mathit{of\text{-}rat\text{-}mult}[\mathit{symmetric}]\ \mathit{of\text{-}rat\text{-}less\text{-}eq}, \mathit{simp}$)
finally
show $\mathit{ge\text{-}0}\ (\mathit{real\text{-}of\text{-}rat}\ p' + \mathit{real\text{-}of\text{-}rat}\ q' * \mathit{sqrt}\ (\mathit{of\text{-}nat}\ b')) =$
$(0 \leq p' \wedge \mathit{of\text{-}nat}\ b' * q' * q' \leq p' * p' \vee 0 \leq q' \wedge p' * p' \leq \mathit{of\text{-}nat}\ b' * q' * q')$
unfolding $\mathit{ge\text{-}0\text{-}def}\ \mathit{p\text{-}def}\ \mathit{b\text{-}def}\ \mathit{q\text{-}def}$
by ($\mathit{auto}\ \mathit{simp}$: $\mathit{of\text{-}rat\text{-}add}\ \mathit{of\text{-}rat\text{-}mult}$)
qed

lemma $\mathit{ma\text{-}0}$: $0 = \mathit{real\text{-}of}\ (\mathit{ma\text{-}of\text{-}rat}\ 0)$ **by** ($\mathit{transfer}, \mathit{auto}$)

lemma $\mathit{ma\text{-}1}$: $1 = \mathit{real\text{-}of}\ (\mathit{ma\text{-}of\text{-}rat}\ 1)$ **by** ($\mathit{transfer}, \mathit{auto}$)

lemma $\mathit{ma\text{-}uminus}$:
$-\ (\mathit{real\text{-}of}\ x) = \mathit{real\text{-}of}\ (\mathit{ma\text{-}uminus}\ x)$
by ($\mathit{transfer}, \mathit{auto}\ \mathit{simp}$: $\mathit{of\text{-}rat\text{-}minus}$)

lemma $\mathit{ma\text{-}inverse}$: $\mathit{inverse}\ (\mathit{real\text{-}of}\ r) = \mathit{real\text{-}of}\ (\mathit{ma\text{-}inverse}\ r)$
proof ($\mathit{transfer}, \mathit{unfold}\ \mathit{Let\text{-}def}, \mathit{clarsimp}$)
fix $p'\ q' :: \mathit{rat}$ **and** $b' :: \mathit{nat}$
assume b': $q' = 0 \vee b' \in \mathit{sqrt\text{-}irrat}$
define b **where** $b = \mathit{real\text{-}of\text{-}nat}\ b'$
define p **where** $p = \mathit{real\text{-}of\text{-}rat}\ p'$
define q **where** $q = \mathit{real\text{-}of\text{-}rat}\ q'$
from b' **have** b: $b \geq 0\ q = 0 \vee b' \in \mathit{sqrt\text{-}irrat}$ **unfolding** $\mathit{b\text{-}def}\ \mathit{q\text{-}def}$ **by** auto
have $\mathit{inverse}\ (p + q * \mathit{sqrt}\ b) = (p - q * \mathit{sqrt}\ b) * \mathit{inverse}\ (p * p - b * (q * q))$
proof ($\mathit{cases}\ q = 0$)
case True **thus** $?\mathit{thesis}$ **by** ($\mathit{cases}\ p = 0, \mathit{auto}\ \mathit{simp}$: $\mathit{field\text{-}simps}$)
next
case False
from $\mathit{sqrt\text{-}irrat}[\mathit{OF}\ b', \mathit{of}\ p']\ \mathit{b\text{-}def}\ \mathit{p\text{-}def}\ \mathit{q\text{-}def}\ \mathit{False}$ **have** nnull: $p + q * \mathit{sqrt}\ b$

```
≠ 0 by auto
    have ?thesis ⟷ (p + q * sqrt b) * inverse (p + q * sqrt b) =
      (p + q * sqrt b) * ((p − q * sqrt b) * inverse (p * p − b * (q * q)))
      unfolding mult-left-cancel[OF nnull] by auto
    also have (p + q * sqrt b) * inverse (p + q * sqrt b) = 1 using nnull by auto
    also have (p + q * sqrt b) * ((p − q * sqrt b) * inverse (p * p − b * (q * q)))
      = (p * p − b * (q * q)) * inverse (p * p − b * (q * q))
      using b by (auto simp: field-simps)
    also have ... = 1
    proof (rule right-inverse, rule)
      assume eq: p * p − b * (q * q) = 0
      have real-of-rat (p' * p' / (q' * q')) = p * p / (q * q)
        unfolding p-def b-def q-def by (auto simp: of-rat-mult of-rat-divide)
      also have ... = b using eq False by (auto simp: field-simps)
      also have ... = real-of-rat (of-nat b') unfolding b-def by auto
      finally have (p' / q') * (p' / q') = of-nat b'
        unfolding of-rat-eq-iff by simp
      with b False show False unfolding sqrt-irrat-def by blast
    qed
    finally
    show ?thesis by simp
  qed
  thus inverse (real-of-rat p' + real-of-rat q' * sqrt (of-nat b')) =
      real-of-rat (p' * inverse (p' * p' − of-nat b' * q' * q')) +
      real-of-rat (− (q' * inverse (p' * p' − of-nat b' * q' * q'))) * sqrt (of-nat b')
   by (simp add: divide-simps of-rat-mult of-rat-divide of-rat-diff of-rat-minus b-def
p-def q-def
          split: if-splits)
qed

lemma ma-sqrt-main: ma-rat r ≥ 0 ⟹ ma-coeff r = 0 ⟹ sqrt (real-of r) =
real-of (ma-sqrt r)
proof (transfer, unfold Let-def, clarsimp)
  fix p :: rat
  assume p: 0 ≤ p
  hence abs: abs p = p by auto
  obtain a b where ab: quotient-of p = (a,b) by force
  hence pab: p = of-int a / of-int b by (rule quotient-of-div)
  from ab have b: b > 0 by (rule quotient-of-denom-pos)
  with p pab have abpos: a * b ≥ 0
    by (metis of-int-0-le-iff of-int-le-0-iff zero-le-divide-iff zero-le-mult-iff)
  have rab: of-nat (nat (a * b)) = real-of-int a * real-of-int b using abpos
    by simp
  let ?lhs = sqrt (of-int a / of-int b)
  let ?rhs = (case case quotient-of p of
            (a, b) ⇒ (case sqrt-int |a * b| of [] ⇒ (0, inverse (of-int b), nat |a * b|)
              | s # x ⇒ (of-int s / of-int b, 0, 0)) of
          (p, q, b) ⇒ of-rat p + of-rat q * sqrt (of-nat b))
  have sqrt (real-of-rat p) = ?lhs unfolding pab
```

```
    by (metis of-rat-divide of-rat-of-int-eq)
  also have ... = ?rhs
  proof (cases sqrt-int |a * b|)
    case Nil
    define sb where sb = sqrt (of-int b)
    define sa where sa = sqrt (of-int a)
    from b sb-def have sb: sb > 0 real-of-int b > 0 by auto
    have sbb: sb * sb = real-of-int b unfolding sb-def
      by (rule sqrt-sqrt, insert b, auto)
    from Nil have ?thesis = (sa / sb =
      inverse (of-int b) * (sa * sb)) unfolding ab sa-def sb-def using abpos
      by (simp add: rab of-rat-divide real-sqrt-mult real-sqrt-divide of-rat-inverse)
    also have ... = (sa = inverse (of-int b) * sa * (sb * sb)) using sb
        by (metis b divide-real-def eq-divide-imp inverse-divide inverse-inverse-eq
inverse-mult-distrib less-int-code(1) of-int-eq-0-iff real-sqrt-eq-zero-cancel-iff sb-def
sbb times-divide-eq-right)
    also have ... = True using sb(2) unfolding sbb by auto
    finally show ?thesis by simp
  next
    case (Cons s x)
    from b have b: real-of-int b > 0 by auto
    from Cons sqrt-int[of abs (a * b)] have s * s = abs (a * b) by auto
    with sqrt-int-pos[OF Cons] have sqrt (real-of-int (abs (a * b))) = of-int s
      by (metis abs-of-nonneg of-int-mult of-int-abs real-sqrt-abs2)
    with abpos have of-int s = sqrt (real-of-int (a * b)) by auto
    thus ?thesis unfolding ab split using Cons b
      by (auto simp: of-rat-divide field-simps real-sqrt-divide real-sqrt-mult)
  qed
  finally show sqrt (real-of-rat p) = ?rhs .
qed

lemma ma-sqrt: sqrt (real-of r) = (if ma-coeff r = 0 then
  (if ma-rat r ≥ 0 then real-of (ma-sqrt r) else − real-of (ma-sqrt (ma-uminus r)))
   else Code.abort (STR ''cannot represent sqrt of irrational number'') (λ -. sqrt
(real-of r)))
proof (cases ma-coeff r = 0)
  case True note 0 = this
  hence 00: ma-coeff (ma-uminus r) = 0 by (transfer, auto)
  show ?thesis
  proof (cases ma-rat r ≥ 0)
    case True
    from ma-sqrt-main[OF this 0] 0 True show ?thesis by auto
  next
    case False
    hence ma-rat (ma-uminus r) ≥ 0 by (transfer, auto)
    from ma-sqrt-main[OF this 00, folded ma-uminus, symmetric] False 0
    show ?thesis by (auto simp: real-sqrt-minus)
  qed
qed auto
```

lemma *ma-plus*:
(*real-of r1* + *real-of r2*) = (*if ma-compatible r1 r2*
then real-of (*ma-plus r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of r1* + *real-of r2*))
by *transfer* (*auto split*: *prod.split simp*: *field-simps of-rat-add*)

lemma *ma-times*:
(*real-of r1* ∗ *real-of r2*) = (*if ma-compatible r1 r2*
then real-of (*ma-times r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of r1* ∗ *real-of r2*))
by *transfer* (*auto split*: *prod.split simp*: *field-simps of-rat-mult of-rat-add*)

lemma *ma-equal*:
HOL.equal (*real-of r1*) (*real-of r2*) = (*if ma-compatible r1 r2*
then ma-equal r1 r2 else
Code.abort (*STR ''different base''*) (λ -. *HOL.equal* (*real-of r1*) (*real-of r2*)))
proof (*transfer*, *unfold equal-real-def*, *clarsimp*)
fix *p1 q1 p2 q2* :: *rat* **and** *b1 b2* :: *nat*
assume *b1*: *q1* = *0* ∨ *b1* ∈ *sqrt-irrat*
assume *b2*: *q2* = *0* ∨ *b2* ∈ *sqrt-irrat*
assume *base*: *q1* = *0* ∨ *q2* = *0* ∨ *b1* = *b2*
let *?l* = *real-of-rat p1* + *real-of-rat q1* ∗ *sqrt* (*of-nat b1*) =
real-of-rat p2 + *real-of-rat q2* ∗ *sqrt* (*of-nat b2*)
let *?m* = *real-of-rat q1* ∗ *sqrt* (*of-nat b1*) = *real-of-rat* (*p2* − *p1*) + *real-of-rat q2* ∗ *sqrt* (*of-nat b2*)
let *?r* = *p1* = *p2* ∧ *q1* = *q2* ∧ (*q1* = *0* ∨ *b1* = *b2*)
have *?l* ⟷ *real-of-rat q1* ∗ *sqrt* (*of-nat b1*) = *real-of-rat* (*p2* − *p1*) + *real-of-rat q2* ∗ *sqrt* (*of-nat b2*)
by (*auto simp*: *of-rat-add of-rat-diff of-rat-minus*)
also have ... ⟷ *p1* = *p2* ∧ *q1* = *q2* ∧ (*q1* = *0* ∨ *b1* = *b2*)
proof
assume *?m*
from *base* **have** *q1* = *0* ∨ *q1* ≠ *0* ∧ *q2* = *0* ∨ *q1* ≠ *0* ∧ *q2* ≠ *0* ∧ *b1* = *b2* **by** *auto*
thus *p1* = *p2* ∧ *q1* = *q2* ∧ (*q1* = *0* ∨ *b1* = *b2*)
proof
assume *q1*: *q1* = *0*
with ‹*?m*› **have** *real-of-rat* (*p2* − *p1*) + *real-of-rat q2* ∗ *sqrt* (*of-nat b2*) = *0* **by** *auto*
with *sqrt-irrat b2* **have** *q2*: *q2* = *0* **by** *auto*
with *q1* ‹*?m*› **show** *?thesis* **by** *auto*
next
assume *q1* ≠ *0* ∧ *q2* = *0* ∨ *q1* ≠ *0* ∧ *q2* ≠ *0* ∧ *b1* = *b2*
thus *?thesis*
proof
assume *ass*: *q1* ≠ *0* ∧ *q2* = *0*
with ‹*?m*› **have** *real-of-rat* (*p1* − *p2*) + *real-of-rat q1* ∗ *sqrt* (*of-nat b1*) = *0*
by (*auto simp*: *of-rat-diff*)

```
      with b1 have q1 = 0 using sqrt-irrat by auto
      with ass show ?thesis by auto
    next
      assume ass: q1 ≠ 0 ∧ q2 ≠ 0 ∧ b1 = b2
     with ‹?m› have *: real-of-rat (p2 − p1) + real-of-rat (q2 − q1) * sqrt (of-nat
b2) = 0
        by (auto simp: field-simps of-rat-diff)
      have q2 − q1 = 0
        by (rule sqrt-irrat[OF - *], insert ass b2, auto)
      with * ass show ?thesis by auto
    qed
  qed
 qed auto
 finally show ?l = ?r by simp
qed

lemma ma-floor: floor (real-of r) = ma-floor r
proof (transfer, unfold Let-def, clarsimp)
 fix p q :: rat and b :: nat
 obtain z1 n1 where qp: quotient-of p = (z1,n1) by force
 obtain z2 n2 where qq: quotient-of q = (z2,n2) by force
 from quotient-of-denom-pos[OF qp] have n1: 0 < n1 .
 from quotient-of-denom-pos[OF qq] have n2: 0 < n2 .
 from quotient-of-div[OF qp] have p: p = of-int z1 / of-int n1 .
 from quotient-of-div[OF qq] have q: q = of-int z2 / of-int n2 .
 have p: p = of-int (z1 * n2) / of-int (n1 * n2) unfolding p using n2 by auto
 have q: q = of-int (z2 * n1) / of-int (n1 * n2) unfolding q using n1 by auto
 define z1n2 where z1n2 = z1 * n2
 define z2n1 where z2n1 = z2 * n1
 define n12 where n12 = n1 * n2
 define r-add where r-add = of-int (z2n1) * sqrt (real-of-int (int b))
 from n1 n2 have n120: n12 > 0 unfolding n12-def by simp
  have floor (of-rat p + of-rat q * sqrt (real-of-nat b)) = floor ((of-int z1n2 +
r-add) / of-int n12)
   unfolding r-add-def n12-def z1n2-def z2n1-def
   unfolding p q add-divide-distrib of-rat-divide of-rat-of-int-eq of-int-of-nat-eq by
simp
 also have ... = floor (of-int z1n2 + r-add) div n12
   by (rule floor-div-pos-int[OF n120])
 also have of-int z1n2 + r-add = r-add + of-int z1n2 by simp
 also have floor (...) = floor r-add + z1n2 by simp
 also have ... = z1n2 + floor r-add by simp
 finally have id: ⌊of-rat p + of-rat q * sqrt (of-nat b)⌋ = (z1n2 + ⌊r-add⌋) div
n12 .
 show ⌊of-rat p + of-rat q * sqrt (of-nat b)⌋ =
          (case quotient-of p of
           (z1, n1) ⇒
             case quotient-of q of
             (z2, n2) ⇒
```

```
                (z1 * n2 + (if 0 ≤ z2 * n1 then sqrt-int-floor-pos (z2 * n1 * (z2 *
n1) * int b) else
                  − sqrt-int-ceiling-pos (z2 * n1 * (z2 * n1) * int b))) div (n1 * n2))
    unfolding qp qq split id n12-def z1n2-def
  proof (rule arg-cong[of - - λ x. ((z1 * n2) + x) div (n1 * n2)])
    have ge-int: z2 * n1 * (z2 * n1) * int b ≥ 0
      by (metis mult-nonneg-nonneg zero-le-square of-nat-0-le-iff)
    show ⌊r-add⌋ = (if 0 ≤ z2 * n1 then sqrt-int-floor-pos (z2 * n1 * (z2 * n1) *
int b) else − sqrt-int-ceiling-pos (z2 * n1 * (z2 * n1) * int b))
    proof (cases z2 * n1 ≥ 0)
      case True
      hence ge: real-of-int (z2 * n1) ≥ 0 by (metis of-int-0-le-iff)
      have radd: r-add = sqrt (of-int (z2 * n1 * (z2 * n1) * int b))
        unfolding r-add-def z2n1-def using sqrt-sqrt[OF ge]
        by (simp add: ac-simps real-sqrt-mult)
       show ?thesis unfolding radd sqrt-int-floor-pos[OF ge-int] using True by
simp
    next
      case False
      hence ge: real-of-int (− (z2 * n1)) ≥ 0
      by (metis mult-zero-left neg-0-le-iff-le of-int-0-le-iff order-refl zero-le-mult-iff)
      have r-add = − sqrt (of-int (z2 * n1 * (z2 * n1) * int b))
        unfolding r-add-def z2n1-def using sqrt-sqrt[OF ge]
        by (metis minus-minus minus-mult-commute minus-mult-right of-int-minus
of-int-mult real-sqrt-minus real-sqrt-mult z2n1-def)
      hence radd: floor r-add = − ceiling (sqrt (of-int (z2 * n1 * (z2 * n1) * int
b)))
        by (metis ceiling-def minus-minus)
      show ?thesis unfolding radd sqrt-int-ceiling-pos[OF ge-int] using False by
simp
    qed
  qed
qed

lemma comparison-impl:
  (x :: real) ≤ (y :: real) = ge-0 (y − x)
  (x :: real) < (y :: real) = (x ≠ y ∧ ge-0 (y − x))
  by (simp-all add: ge-0-def, linarith+)

lemma ma-of-rat: real-of-rat r = real-of (ma-of-rat r)
  by (transfer, auto)

definition is-rat :: real ⇒ bool where
  [code-abbrev]: is-rat x ⟷ x ∈ ℚ

lemma ma-is-rat: is-rat (real-of x) = ma-is-rat x
proof (transfer, unfold is-rat-def, clarsimp)
  fix p q :: rat and b :: nat
  let ?p = real-of-rat p
```

```
  let ?q = real-of-rat q
  let ?b = real-of-nat b
  let ?b' = real-of-rat (of-nat b)
  assume b: q = 0 ∨ b ∈ sqrt-irrat
  show (?p + ?q * sqrt ?b ∈ ℚ) = (q = 0)
  proof (cases q = 0)
    case False
    from False b have b: b ∈ sqrt-irrat by auto
    {
      assume ?p + ?q * sqrt ?b ∈ ℚ
     from this[unfolded Rats-def] obtain r where r: ?p + ?q * sqrt ?b = real-of-rat
r by auto
      let ?r = real-of-rat r
     from r have real-of-rat (p − r) + ?q * sqrt ?b = 0 by (simp add: of-rat-diff)
      from sqrt-irrat[OF disjI2[OF b] this] False have False by auto
    }
    thus ?thesis by auto
  qed auto
qed


definition sqrt-real x = (if x ∈ ℚ ∧ x ≥ 0 then (if x = 0 then [0] else (let sx =
sqrt x in [sx,−sx])) else [])

lemma sqrt-real[simp]: assumes x: x ∈ ℚ
  shows set (sqrt-real x) = {y . y * y = x}
proof (cases x ≥ 0)
  case False
  hence ⋀ y. y * y ≠ x by auto
  with False show ?thesis unfolding sqrt-real-def by auto
next
  case True
  thus ?thesis using x
    by (cases x = 0, auto simp: Let-def sqrt-real-def)
qed

code-datatype real-of

declare [[code drop:
  plus :: real ⇒ real ⇒ real
  uminus :: real ⇒ real
  times :: real ⇒ real ⇒ real
  inverse :: real ⇒ real
  floor :: real ⇒ int
  sqrt
  HOL.equal :: real ⇒ real ⇒ bool
]]

lemma [code]:
```

Ratreal = *real-of* ∘ *ma-of-rat*
by (*simp add*: *fun-eq-iff*) (*transfer*, *simp*)

lemmas *ma-code-eqns* [*code equation*] = *ma-ge-0 ma-floor ma-0 ma-1 ma-uminus ma-inverse ma-sqrt ma-plus ma-times ma-equal ma-is-rat*
comparison-impl

lemma [*code equation*]:
(*x* :: *real*) / (*y* :: *real*) = *x* ∗ *inverse y*
(*x* :: *real*) − (*y* :: *real*) = *x* + (− *y*)
by (*simp-all add*: *divide-inverse*)

Some tests with small numbers. To work on larger number, one should additionally import the theories for efficient calculation on numbers

value ⌊*101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*)⌋
value ⌊*606.2* ∗ *sqrt 2* + *0.001*⌋
value *101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*) = *606.2* ∗ *sqrt 2* + *0.001*
value *101.1* ∗ (*3* ∗ *sqrt 2* + *6* ∗ *sqrt 0.5*) > *606.2* ∗ *sqrt 2* + *0.001*
value (*sqrt 0.1* ∈ ℚ, *sqrt* (− *0.09*) ∈ ℚ)

end

5 A unique representation of real numbers via triples

theory *Real-Unique-Impl*
imports
Prime-Product
Real-Impl
Show.Show-Instances
Show.Show-Real
begin

We implement the real numbers again using triples, but now we require an additional invariant on the triples, namely that the base has to be a prime product. This has the consequence that the mapping of triples into ℝ is injective. Hence, equality on reals is now equality on triples, which can even be executed in case of different bases. Similarly, we now also allow different basis in comparisons. Ultimately, injectivity allows us to define a show-function for real numbers, which pretty prints real numbers into strings.

typedef *mini-alg-unique* =
{ *r* :: *mini-alg* . *ma-coeff r* = *0* ∧ *ma-base r* = *0* ∨ *ma-coeff r* ≠ *0* ∧ *prime-product* (*ma-base r*)}
by (*transfer*, *auto*)

setup-lifting *type-definition-mini-alg-unique*

lift-definition *real-of-u* :: *mini-alg-unique* ⇒ *real* **is** *real-of* **.**

lift-definition *mau-floor* :: *mini-alg-unique* ⇒ *int* **is** *ma-floor* .
lift-definition *mau-of-rat* :: *rat* ⇒ *mini-alg-unique* **is** *ma-of-rat* **by** (*transfer*, *auto*)
lift-definition *mau-rat* :: *mini-alg-unique* ⇒ *rat* **is** *ma-rat* .
lift-definition *mau-base* :: *mini-alg-unique* ⇒ *nat* **is** *ma-base* .
lift-definition *mau-coeff* :: *mini-alg-unique* ⇒ *rat* **is** *ma-coeff* .
lift-definition *mau-uminus* :: *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-uminus*
by (*transfer*, *auto*)
lift-definition *mau-compatible* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *bool* **is**
ma-compatible .
lift-definition *mau-ge-0* :: *mini-alg-unique* ⇒ *bool* **is** *ma-ge-0* .
lift-definition *mau-inverse* :: *mini-alg-unique* ⇒ *mini-alg-unique* **is** *ma-inverse*
by (*transfer*, *auto simp*: *ma-normalize-def Let-def split*: *if-splits*)
lift-definition *mau-plus* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *mini-alg-unique*
is *ma-plus*
by (*transfer*, *auto simp*: *ma-normalize-def split*: *if-splits*)
lift-definition *mau-times* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *mini-alg-unique*
is *ma-times*
by (*transfer*, *auto simp*: *ma-normalize-def split*: *if-splits*)
lift-definition *ma-identity* :: *mini-alg* ⇒ *mini-alg* ⇒ *bool* **is** (=) .
lift-definition *mau-equal* :: *mini-alg-unique* ⇒ *mini-alg-unique* ⇒ *bool* **is** *ma-identity*
.
lift-definition *mau-is-rat* :: *mini-alg-unique* ⇒ *bool* **is** *ma-is-rat* .

lemma *Ratreal-code*[*code*]:
Ratreal = *real-of-u* ∘ *mau-of-rat*
by (*simp add*: *fun-eq-iff*) (*transfer*, *transfer*, *simp*)

lemma *mau-floor*: *floor* (*real-of-u r*) = *mau-floor r*
using *ma-floor* **by** (*transfer*, *auto*)
lemma *mau-inverse*: *inverse* (*real-of-u r*) = *real-of-u* (*mau-inverse r*)
using *ma-inverse* **by** (*transfer*, *auto*)
lemma *mau-uminus*: − (*real-of-u r*) = *real-of-u* (*mau-uminus r*)
using *ma-uminus* **by** (*transfer*, *auto*)
lemma *mau-times*:
(*real-of-u r1* ∗ *real-of-u r2*) = (*if mau-compatible r1 r2*
then real-of-u (*mau-times r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of-u r1* ∗ *real-of-u r2*))
using *ma-times* **by** (*transfer*, *auto*)
lemma *mau-plus*:
(*real-of-u r1* + *real-of-u r2*) = (*if mau-compatible r1 r2*
then real-of-u (*mau-plus r1 r2*) *else*
Code.abort (*STR ''different base''*) (λ -. *real-of-u r1* + *real-of-u r2*))
using *ma-plus* **by** (*transfer*, *auto*)

lemma *real-of-u-inj*[*simp*]: *real-of-u x* = *real-of-u y* ⟷ *x* = *y*
proof
note *field-simps*[*simp*] *of-rat-diff*[*simp*]
assume *real-of-u x* = *real-of-u y*
thus *x* = *y*

```
  proof (transfer)
    fix x y
    assume ma-coeff x = 0 ∧ ma-base x = 0 ∨ ma-coeff x ≠ 0 ∧ prime-product
(ma-base x)
       and  ma-coeff y = 0 ∧ ma-base y = 0 ∨ ma-coeff y ≠ 0 ∧ prime-product
(ma-base y)
       and  real-of x = real-of y
    thus x = y
    proof (transfer, clarsimp)
      fix p1 q1 p2 q2 :: rat and b1 b2
      let ?p1 = real-of-rat p1
      let ?p2 = real-of-rat p2
      let ?q1 = real-of-rat q1
      let ?q2 = real-of-rat q2
      let ?b1 = real-of-nat b1
      let ?b2 = real-of-nat b2
      assume q1: q1 = 0 ∧ b1 = 0 ∨ q1 ≠ 0 ∧ prime-product b1
      and q2: q2 = 0 ∧ b2 = 0 ∨ q2 ≠ 0 ∧ prime-product b2
      and i1: q1 = 0 ∨ b1 ∈ sqrt-irrat
      and i2: q2 = 0 ∨ b2 ∈ sqrt-irrat
      and eq: ?p1 + ?q1 * sqrt ?b1 = ?p2 + ?q2 * sqrt ?b2
      show p1 = p2 ∧ q1 = q2 ∧ b1 = b2
      proof (cases q1 = 0)
        case True
        have q2 = 0
          by (rule sqrt-irrat[OF i2, of p2 − p1], insert eq True q1, auto)
        with True q1 q2 eq show ?thesis by auto
      next
        case False
        hence 1: q1 ≠ 0 prime-product b1 using q1 by auto
        {
          assume ∗: q2 = 0
          have q1 = 0
            by (rule sqrt-irrat[OF i1, of p1 − p2], insert eq ∗ q2, auto)
          with False have False by auto
        }
        hence 2: q2 ≠ 0 prime-product b2 using q2 by auto
        from 1 i1 have b1: b1 ≠ 0 unfolding sqrt-irrat-def by (cases b1, auto)
        from 2 i2 have b2: b2 ≠ 0 unfolding sqrt-irrat-def by (cases b2, auto)
        let ?sq = λ x. x ∗ x
        define q3 where q3 = p2 − p1
        let ?q3 = real-of-rat q3
       let ?e = of-rat (q2 ∗ q2 ∗ of-nat b2 + ?sq q3  − ?sq q1 ∗ of-nat b1) + of-rat
(2 ∗ q2 ∗ q3) ∗ sqrt ?b2
        from eq have ∗: ?q1 ∗ sqrt ?b1 = ?q2 ∗ sqrt ?b2 + ?q3
          by (simp add: q3-def)
        from arg-cong[OF this, of ?sq] have 0 = (real-of-rat 2 ∗ ?q2 ∗ ?q3) ∗ sqrt
?b2 +
          (?sq ?q2 ∗ ?b2 +  ?sq ?q3 − ?sq ?q1 ∗ ?b1)
```

```
    by auto
  also have ... = ?e
    by (simp add: of-rat-mult of-rat-add of-rat-minus)
  finally have eq: ?e = 0 by simp
  from sqrt-irrat[OF - this] 2 i2 have q3: q3 = 0 by auto
  hence p: p1 = p2 unfolding q3-def by simp
  let ?b1 = rat-of-nat b1
  let ?b2 = rat-of-nat b2
  from eq[unfolded q3] have eq: ?sq q2 * ?b2 = ?sq q1 * ?b1 by auto
  obtain z1 n1 where d1: quotient-of q1 = (z1,n1) by force
  obtain z2 n2 where d2: quotient-of q2 = (z2,n2) by force
  note id = quotient-of-div[OF d1] quotient-of-div[OF d2]
  note pos = quotient-of-denom-pos[OF d1] quotient-of-denom-pos[OF d2]
  from id(1) 1(1) pos(1) have z1: z1 ≠ 0 by auto
  from id(2) 2(1) pos(2) have z2: z2 ≠ 0 by auto
  let ?n1 = rat-of-int n1
  let ?n2 = rat-of-int n2
  let ?z1 = rat-of-int z1
  let ?z2 = rat-of-int z2
  from arg-cong[OF eq[simplified id], of λ x. x * ?sq ?n1 * ?sq ?n2,
    simplified field-simps]
  have ?sq (?n1 * ?z2) * ?b2 = ?sq (?n2 * ?z1) * ?b1
    using pos by auto
  moreover have ?n1 * ?z2 ≠ 0 ?n2 * ?z1 ≠ 0 using z1 z2 pos by auto
  ultimately obtain i1 i2 where 0: rat-of-int i1 ≠ 0 rat-of-int i2 ≠ 0
    and eq: ?sq (rat-of-int i2) * ?b2 = ?sq (rat-of-int i1) * ?b1
    unfolding of-int-mult[symmetric] by blast+
  let ?b1 = int b1
  let ?b2 = int b2
  from eq have eq: ?sq i1 * ?b1 = ?sq i2 * ?b2
    by (metis (hide-lams, no-types) of-int-eq-iff of-int-mult of-int-of-nat-eq)
  from 0 have 0: i1 ≠ 0 i2 ≠ 0 by auto
  from arg-cong[OF eq, of nat] have ?sq (nat (abs i1)) * b1 = ?sq (nat (abs
i2)) * b2
    by (metis abs-of-nat eq nat-abs-mult-distrib nat-int)
  moreover have nat (abs i1) > 0 nat (abs i2) > 0 using 0 by auto
  ultimately obtain n1 n2 where 0: n1 > 0 n2 > 0 and eq: ?sq n1 * b1 =
?sq n2 * b2 by blast
  from b1 0 have b1: b1 > 0 n1 > 0 n1 * n1 > 0 by auto
  from b2 0 have b2: b2 > 0 n2 > 0 n2 * n2 > 0 by auto
  {
    fix p :: nat assume p: prime p
    have multiplicity p (?sq n1 * b1) = multiplicity p b1 + 2 * multiplicity p
n1
      using b1 p by (auto simp: prime-elem-multiplicity-mult-distrib)
    also have ... mod 2 = multiplicity p b1 mod 2 by presburger
    finally have id1: multiplicity p (?sq n1 * b1) mod 2 = multiplicity p b1
mod 2 .
    have multiplicity p (?sq n2 * b2) = multiplicity p b2 + 2 * multiplicity p
```

```
n2
          using b2 p by (auto simp: prime-elem-multiplicity-mult-distrib)
        also have ... mod 2 = multiplicity p b2 mod 2 by presburger
        finally have id2: multiplicity p (?sq n2 * b2) mod 2 = multiplicity p b2
mod 2 .
        from id1 id2 eq have eq: multiplicity p b1 mod 2 = multiplicity p b2 mod
2 by simp
        from 1(2) 2(2) p have multiplicity p b1 ≤ 1 multiplicity p b2 ≤ 1
          unfolding prime-product-def by auto
        with eq have multiplicity p b1 = multiplicity p b2 by simp
      }
      with b1(1) b2(1) have b: b1 = b2 by (rule multiplicity-eq-nat)
      from *[unfolded b q3] b1(1) b2(1) have q: q1 = q2 by simp
      from p q b show ?thesis by blast
    qed
  qed
 qed
qed simp

lift-definition mau-sqrt :: mini-alg-unique ⇒ mini-alg-unique is
  λ ma. let (a,b) = quotient-of (ma-rat ma); (sq,fact) = prime-product-factor (nat
(abs a * b));
      ma' = ma-of-rat (of-int (sgn(a)) * of-nat sq / of-int b)
      in ma-times ma' (ma-sqrt (ma-of-rat (of-nat fact)))
proof −
 fix ma :: mini-alg
 let ?num =
   let (a, b) = quotient-of (ma-rat ma); (sq, fact) = prime-product-factor (nat (|a|
* b));
       ma' = ma-of-rat (rat-of-int (sgn a) * rat-of-nat sq / of-int b)
     in ma-times ma' (ma-sqrt (ma-of-rat (rat-of-nat fact)))
 obtain a b where q: quotient-of (ma-rat ma) = (a,b) by force
 obtain sq fact where ppf: prime-product-factor (nat (abs a * b)) = (sq,fact) by
force
 define asq where asq = rat-of-int (sgn a) * of-nat sq / of-int b
 define ma' where ma' = ma-of-rat asq
 define sqrt where sqrt = ma-sqrt (ma-of-rat (rat-of-nat fact))
  have num: ?num = ma-times ma' sqrt unfolding q ppf asq-def Let-def split
ma'-def sqrt-def ..
 let ?inv = λ ma. ma-coeff ma = 0 ∧ ma-base ma = 0 ∨ ma-coeff ma ≠ 0 ∧
prime-product (ma-base ma)
 have ma': ?inv ma' unfolding ma'-def
   by (transfer, auto)
 have id: ⋀ i. int i * 1 = i ⋀ i :: rat. i / 1 = i rat-of-int 1 = 1 inverse (1 :: rat)
= 1
   ⋀ n. nat |int n| = n by auto
 from prime-product-factor[OF ppf] have prime-product fact by auto
 hence sqrt: ?inv sqrt unfolding sqrt-def
   by (transfer, unfold split quotient-of-nat Let-def id, case-tac sqrt-int |int facta|,
```

auto)
show *?inv ?num* **unfolding** *num* **using** *ma' sqrt*
by (*transfer*, *auto simp*: *ma-normalize-def split*: *if-splits*)
qed

lemma *sqrt-sgn*[*simp*]: *sqrt* (*of-int* (*sgn a*)) = *of-int* (*sgn a*)
by (*cases a* ≥ *0*, *cases a* = *0*, *auto simp*: *real-sqrt-minus*)

lemma *mau-sqrt-main*: *mau-coeff r* = *0* ⟹ *sqrt* (*real-of-u r*) = *real-of-u* (*mau-sqrt r*)
proof (*transfer*)
fix *r*
assume *ma-coeff r* = *0*
hence *rr*: *real-of r* = *of-rat* (*ma-rat r*) **by** (*transfer*, *auto*)
obtain *a b* **where** *q*: *quotient-of* (*ma-rat r*) = (*a*,*b*) **by** *force*
from *quotient-of-div*[*OF q*] **have** *r*: *ma-rat r* = *of-int a* / *of-int b* **by** *auto*
from *quotient-of-denom-pos*[*OF q*] **have** *b*: *b* > *0* **by** *auto*
obtain *sq fact* **where** *ppf*: *prime-product-factor* (*nat* (|*a*| ∗ *b*)) = (*sq*, *fact*) **by** *force*
from *prime-product-factor*[*OF ppf*] **have** *ab*: *nat* (|*a*| ∗ *b*) = *sq* ∗ *sq* ∗ *fact* **..**
have *sqrt* (*real-of r*) = *sqrt*(*of-int a* / *of-int b*) **unfolding** *rr r*
by (*metis of-rat-divide of-rat-of-int-eq*)
also have *real-of-int a* / *of-int b* = *of-int a* ∗ *of-int b* / (*of-int b* ∗ *of-int b*) **using** *b* **by** *auto*
also have *sqrt* (...) = *sqrt* (*of-int a* ∗ *of-int b*) / *of-int b* **using** *sqrt-sqrt*[*of real-of-int b*] *b*
by (*metis less-eq-real-def of-int-0-less-iff real-sqrt-divide real-sqrt-mult*)
also have *real-of-int a* ∗ *of-int b* = *real-of-int* (*a* ∗ *b*) **by** *auto*
also have *a* ∗ *b* = *sgn a* ∗ (*abs a* ∗ *b*) **by** (*simp*, *metis mult-sgn-abs*)
also have *real-of-int* (...) = *of-int* (*sgn a*) ∗ *real-of-int* (|*a*| ∗ *b*)
unfolding *of-int-mult*[*of sgn a*] **..**
also have *real-of-int* (|*a*| ∗ *b*) = *of-nat* (*nat* (*abs a* ∗ *b*)) **using** *b*
by (*metis abs-sgn mult-pos-pos mult-zero-left nat-int of-int-of-nat-eq of-nat-0 zero-less-abs-iff zero-less-imp-eq-int*)
also have ... = *of-nat fact* ∗ (*of-nat sq* ∗ *of-nat sq*) **unfolding** *ab of-nat-mult* **by** *simp*
also have *sqrt* (*of-int* (*sgn a*) ∗ (*of-nat fact* ∗ (*of-nat sq* ∗ *of-nat sq*))) =
of-int (*sgn a*) ∗ *sqrt* (*of-nat fact*) ∗ *of-nat sq*
unfolding *real-sqrt-mult* **by** *simp*
finally have *r*: *sqrt* (*real-of r*) = *real-of-int* (*sgn a*) ∗ *real-of-nat sq* / *real-of-int b* ∗ *sqrt* (*real-of-nat fact*) **by** *simp*
let *?asqb* = *ma-of-rat* (*rat-of-int* (*sgn a*) ∗ *rat-of-nat sq* / *rat-of-int b*)
let *?f* = *ma-of-rat* (*rat-of-nat fact*)
let *?sq* = *ma-sqrt ?f*
have *sq*: *0* ≤ *ma-rat ?f ma-coeff ?f* = *0* **by** ((*transfer*, *simp*)+)
have *compat*: ⋀ *m*. (*ma-compatible ?asqb m*) = *True*
by (*transfer*, *auto*)
show *sqrt* (*real-of r*) =
real-of

```
        (let (a, b) = quotient-of (ma-rat r); (sq, fact) = prime-product-factor (nat
(|a| * b));
             ma' = ma-of-rat (rat-of-int (sgn a) * rat-of-nat sq / rat-of-int b)
         in ma-times ma' (ma-sqrt (ma-of-rat (rat-of-nat fact))))
    unfolding q ppf Let-def split
    unfolding r
    unfolding ma-times[symmetric, of ?asqb, unfolded compat if-True]
    unfolding ma-sqrt-main[OF sq, symmetric]
    unfolding ma-of-rat[symmetric]
    by (simp add: of-rat-divide of-rat-mult)
qed

lemma mau-sqrt: sqrt (real-of-u r) = (if mau-coeff r = 0 then
  real-of-u (mau-sqrt r)
   else Code.abort (STR ''cannot represent sqrt of irrational number'') (λ -. sqrt
(real-of-u r)))
  using mau-sqrt-main[of r] by (cases mau-coeff r = 0, auto)

lemma mau-0: 0 = real-of-u (mau-of-rat 0) using ma-0 by (transfer, auto)

lemma mau-1: 1 = real-of-u (mau-of-rat 1) using ma-1 by (transfer, auto)

lemma mau-equal:
 HOL.equal (real-of-u r1) (real-of-u r2) = mau-equal r1 r2 unfolding equal-real-def
  using real-of-u-inj[of r1 r2]
  by (transfer, transfer, auto)

lemma mau-ge-0: ge-0 (real-of-u x) = mau-ge-0 x using ma-ge-0
  by (transfer, auto)

definition real-lt :: real ⇒ real ⇒ bool where real-lt = (<)
```

The following code equation terminates if it is started on two different inputs.

```
lemma real-lt [code equation]: real-lt x y = (let fx = floor x; fy = floor y in
  (if fx < fy then True else if fx > fy then False else real-lt (x * 1024) (y * 1024)))
proof (cases floor x < floor y)
  case True
  thus ?thesis by (auto simp: real-lt-def floor-less-cancel)
next
  case False note nless = this
  show ?thesis
  proof (cases floor x > floor y)
    case True
    from floor-less-cancel[OF this] True nless show ?thesis
      by (simp add: real-lt-def)
  next
    case False
    with nless show ?thesis unfolding real-lt-def by auto
```

qed
qed

For comparisons we first check for equality. Then, if the bases are compatible we can just compare the differences with 0. Otherwise, we start the recursive algorithm *real-lt* which works on arbitrary bases. In this way, we have an implementation of comparisons which can compare all representable numbers.

Note that in *Real-Impl.Real-Impl* we did not use *real-lt* as there the code-equations for equality already require identical bases.

lemma *comparison-impl*:
real-of-u x ≤ *real-of-u y* ⟷ *real-of-u x* = *real-of-u y* ∨
(*if mau-compatible x y then ge-0* (*real-of-u y* − *real-of-u x*) *else real-lt* (*real-of-u x*) (*real-of-u y*))
real-of-u x < *real-of-u y* ⟷ *real-of-u x* ≠ *real-of-u y* ∧
(*if mau-compatible x y then ge-0* (*real-of-u y* − *real-of-u x*) *else real-lt* (*real-of-u x*) (*real-of-u y*))
unfolding *ge-0-def real-lt-def* **by** (*auto simp del*: *real-of-u-inj*)

lemma *mau-is-rat*: *is-rat* (*real-of-u x*) = *mau-is-rat x* **using** *ma-is-rat*
by (*transfer*, *auto*)

lift-definition *ma-show-real* :: *mini-alg* ⇒ *string* **is**
λ (*p*,*q*,*b*). *let sb* = *shows* ''*sqrt*('' ∘ *shows b* ∘ *shows* '')'';
qb = (*if q* = *1 then sb else if q* = −*1 then shows* ''−'' ∘ *sb else shows q* ∘ *shows* ''∗'' ∘ *sb*) *in*
if q = *0 then shows p* [] *else*
if p = *0 then qb* [] *else*
if q < *0 then* ((*shows p* ∘ *qb*) [])
else ((*shows p* ∘ *shows* ''+'' ∘ *qb*) []) **.**

lift-definition *mau-show-real* :: *mini-alg-unique* ⇒ *string* **is** *ma-show-real* **.**

overloading *show-real* ≡ *show-real*
begin
definition *show-real*
where *show-real x* ≡
(*if* (∃ *y*. *x* = *real-of-u y*) *then mau-show-real* (*THE y*. *x* = *real-of-u y*) *else* [])
end

lemma *mau-show-real*: *show-real* (*real-of-u x*) = *mau-show-real x*
unfolding *show-real-def* **by** *simp*

code-datatype *real-of-u*

declare [[*code drop*:
plus :: *real* ⇒ *real* ⇒ *real*
uminus :: *real* ⇒ *real*
times :: *real* ⇒ *real* ⇒ *real*

inverse :: *real* ⇒ *real*
floor :: *real* ⇒ *int*
sqrt
HOL.equal :: *real* ⇒ *real* ⇒ *bool*
ge-0
is-rat
less :: *real* ⇒ *real* ⇒ *bool*
less-eq :: *real* ⇒ *real* ⇒ *bool*
]]

lemmas *mau-code-eqns* [*code*] = *mau-floor mau-0 mau-1 mau-uminus mau-inverse mau-sqrt mau-plus mau-times mau-equal mau-ge-0 mau-is-rat*
mau-show-real comparison-impl

Some tests with small numbers. To work on larger number, one should additionally import the theories for efficient calculation on numbers

value ⌊*101.1* ∗ (*sqrt 18* + *6* ∗ *sqrt 0.5*)⌋
value ⌊*324* ∗ *sqrt 7* + *0.001*⌋
value *101.1* ∗ (*sqrt 18* + *6* ∗ *sqrt 0.5*) = *324* ∗ *sqrt 7* + *0.001*
value *101.1* ∗ (*sqrt 18* + *6* ∗ *sqrt 0.5*) > *324* ∗ *sqrt 7* + *0.001*
value *show* (*101.1* ∗ (*sqrt 18* + *6* ∗ *sqrt 0.5*))
value (*sqrt 0.1* ∈ ℚ, *sqrt* (− *0.09*) ∈ ℚ)

end

Acknowledgements

We thank Bertram Felgenhauer for interesting discussions and especially for mentioning Cauchy's mean theorem during the formalization of the algorithms for computing roots.

References

[1] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 100–115, 2013.

[2] A. Lochbihler. Light-weight containers for Isabelle: Efficient, extensible, nestable. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 116–132, 2013.

[3] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[4] S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Appl. Algebra in Engineering, Communication and Computing*, 17(1):49–73, 2006.

[5] S. Lucas. Practical use of polynomials over the reals in proofs of termination. In *Principles and Practice of Declarative Programming*, pages 39–50, 2007.

[6] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Theorem Proving in Higher Order Logics*, LNCS 5674, pages 452–468, 2009.

[7] H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Logic for Programming and Automated Reasoning*, volume 6355, pages 481–500, 2010.

[8] H. Zankl, R. Thiemann, and A. Middeldorp. Satisfiability of non-linear arithmetic over algebraic numbers. In *Symbolic Computation in Software Science*, volume 10-10 of *RISC-Linz Technical Report*, pages 19–24, 2010.